



12

**EUROPEAN PATENT APPLICATION**

21 Application number: 93117333.0

51 Int. Cl.<sup>5</sup>: **G06F 9/44**

22 Date of filing: **26.10.93**

30 Priority: 09.11.92 US 972779

43 Date of publication of application:  
**18.05.94 Bulletin 94/20**

54 Designated Contracting States:  
**AT CH DE FR GB IT LI NL SE**

71 Applicant: **VIRTUAL PROTOTYPES, INC.**  
**5252 De Maisonneuve West, Suite 318**  
**Montreal, Quebec H4A 3S5 (CA)**

72 Inventor: **Joseph, Eugene R.**  
**4682 Roslyn Street**  
**Montreal (Quebec) (CA)**  
Inventor: **Trachtman, Michael**  
**542 California Street**  
**Newton, Massachusetts 02160 (US)**

73 Representative: **Zeitler & Dickel**  
**Patentanwälte European Patent Attorneys**  
**Postfach 26 02 51**  
**D-80059 München (DE)**

54 **Computer simulation system and method for specifying the behavior of graphical operator interfaces.**

57 A computer simulator system allows the user to specify prototype reaction to events in a pictorial manner using a visual object environment. A spreadsheet like State Table and a visual object collection coincide on a common graphical display. The State Table is filled in by pointing to lists of events or actions associated with the different objects. The contents of these lists are dependent on the object class. Event or action descriptions are transported into the respective cells of the State Table in the form of descriptive strings of text. This text describes the event or action, and the event source, or action destination. Entries in the State Table define the operation of the simulation and are executed directly by an interpreter or are compiled to generate a program of instructions for performing the simulation.

## BACKGROUND OF THE INVENTION

## 1. Field of the Invention

The invention relates to a system for and method of interactive, graphical computer simulation and, in particular, to a system and method for providing user programming of interactive, graphic and dynamic applications through pictorial means.

## 2. Description of the Related Art

Digital computers have long been used to perform complicated and tedious numeric calculations and data processing. With increased processing speed and memory capability, the roles of digital computers have expanded to include real-time simulations of mechanical, electrical, and other complex systems.

At first, software for performing digital computer simulation required advanced programming skills, the programmer coding the instructions in a low level language such as assembly language or machine code. This was done to provide the necessary interface between operator inputs and graphic objects displayed on an output device such as a graphic display or video monitor. With the development of high level languages, it became feasible to construct program generator systems to assist in the development of programs for performing the simulation. An example of such a simulation system is the Virtual Applications Prototyping System (VAPS) commercially available from the assignee of the subject application. The system is disclosed in system documentation publications including "VAPS: Conceptual Overview"; "VAPS: Reference Guide"; "VAPS: Programmer's Guide"; and "VAPS: Installation Guide and Options" all by Virtual Prototypes (1991) and incorporated herein by reference.

The VAPS system is a tool for designing, testing, and implementing user interfaces. A prototype or "modeled" system is the result of the design work done with design tools constituting the VAPS simulation system. A user interface is a part of hardware and software that provides for human control and feedback. For a software program, human control and feedback can include graphs, menus and prompts. For a product such as a radio, the interface can include scales, dials, and knobs. Thus, the user interface is a bridge for a human to access an application.

The resultant virtualized "prototype" system is a graphics simulation of a control and display mechanism equivalent in appearance, form, functionality, behavior, animation, etc. to the modeled system. The virtualized prototype representation is also applicable to the generation of source code implementing equivalent functionality on a target system. Thus, as used herein, the prototype system is functionally equivalent to a modeled system. However, while the virtualized system is termed to be a prototype, it is not limited to initial system development but may constitute the final target system.

The VAPS system supports development and testing of dynamic, graphical front ends for existing applications and maintenance of the user interface by allowing changes to be made visually. VAPS contains four primary modules or subsystems: (i) an Object Editor, (ii) an Integration Editor, (iii) a Logic Editor, and (iv) a Runtime Environment.

Briefly, the Object Editor is used by the operator to build the visual interface to the application. Using the Object Editor, the user draws objects that make up the graphical interface for the prototype. The objects are then assigned runtime behavior. The user employs the Integration Editor to define and build data connections or routes between the visual interface and an Application program data. That is, after the display image is produced using the Object Editor, the Integration Editor connects objects to each other, to data sinks, and to data sources. At the system level, data connections can be made by defining global prototype/applications system variables.

The Logic Editor allows the user to define and specify how events are to be managed. Using the Logic Editor, the user can develop a logic control program to manage events from the user interface. Finally, the Runtime Environment animates the prototype and allows the user to interact with the prototype.

In the VAPS system, the Object Editor is a window-based, graphical system used to draw and describe objects that make up a man-machine interface (MMI). The VAPS MMIs are collections of objects having a visual representation created by the user and a functional behavior assigned to the graphical representation.

The Object Editor allows the user to build the "functional behavior" exhibited by complex objects or displays in runtime by specifying a list of object descriptions. The functional behavior of an object refers to how the graphic representation of an object responds to data received and generated by the prototype.

The object descriptions define two types of objects: input and output objects. In general, input objects react to user control to supply data to the simulation program representing the "prototype" or "modeled" system. Output objects react to data generated by the input objects, as well as to internal and external data sources

of data. Output objects reflect the data by changing appearance in accordance with the predefined functional behavior of the respective object type. Thus, an output object such as a virtual meter might be graphically represented by a graduated scale and a pointer, the position of the pointer relative to the scale being responsive to predefined data generated by the prototype system.

Input objects accept user inputs and pass data to the system for processing. Typical classes of virtual input objects include buttons, locators, menus, knobs, switches, event labels, input fields, etc. Virtual input objects can be defined to be responsive to a corresponding physical input device, such as a mouse, keyboard, touch screen validator, or a custom device.

Once the application program has processed the information, the results are displayed on a video monitor screen using output objects. Typical classes of output objects include dials, lights, plots, cursors, bar charts, tapes, output fields, etc. Thus, output objects are different representations enabling viewing of data generated by the application. When the data changes, the output objects are dynamically modified to reflect the new value.

The functionality, i.e. behavior, of objects is defined using property sheets. The contents of a property sheet are dependent on the class of the object described by the property sheet. For example, a property sheet for a dial class (i.e., rotational) object defines the name of the object, display position origin, moving part, center of rotation, position pointer, initial pointer position, direction of motion, type of motion, and minimum and maximum positions and corresponding values.

As described, the Object Editor is used to draw the visual portion of the simulated prototype and define the functionality of objects. The Integration Editor Subsystem animates the objects by providing a data connection to the application data. Connections with the Integration Editor are made by specifying a correspondence or mapping between an object and a data source or data sink.

For example, if a "Dial" type of output object represents the temperature in a boiler, the temperature can be changed by an external source. The correspondence between the temperature and the Dial reading is made in the Integration Editor. Similarly, an input object that is created and functionally defined in the Object Editor is connected via globally defined data to provide user input to the application program.

Integration produces a data connection allowing virtual input objects to transmit data and output objects to receive data. The data controls the functional behavior of a virtual object or alters the attributes of a virtual object, such as a displayed position or shape of the virtual object.

Using the Integration Editor, data connections to and from a virtual object are made through a data channel. Data connections between virtual objects can be made directly from one virtual object to a series of other virtual objects. This is accomplished by storing the variable information from an input virtual object in a memory location, referred to as a data channel, and connecting the one or more output virtual objects to the same memory location to read the updated variable.

Data generated by data generators of the prototype interface are connected through direct point-and-click connections. Thus, an output virtual objects can directly receive data generated by the MMI. As used herein, "point-and-click" and "click on" refer to designation of an element or virtual object displayed on a video monitor. This is typically accomplished using a mouse, joystick, touch screen or other graphic input device to position a cursor displayed on the monitor (i.e., point) and activating a momentary switch to designate the element or virtual object at the cursor position (i.e., click or click-on) for further processing.

External user-defined application programs can also provide data. The memory locations for input and output virtual objects can be configured to transfer data to and from an external user application program, with the application program reading and writing to predefined data channels.

Finally, real devices can be directly integrated into the prototype with the interface. Such real devices include input devices such as sensors, joysticks, trackballs, and keyboards.

The Integration Editor maps user input and output information to data locations without regard to where the data originates. A pictorial interface maps data to provide a direct outlet to a virtual object called a plug. All virtual objects have such an associated plug, i.e., a labeled data location accessible under program control, to make the data available to virtual objects and applications programs. In comparison with having to program memory connections through a programming language, mapping using the Integration Editor allows the user to define how the virtual objects of the prototype are to communicate with each other by passing data back and forth using plugs.

Each mapping variable is defined by making a pictorial connection between the plug and the data source or sink associated with a virtual object. Therefore, when a virtual input objects of a prototype are connected to other data sources or sinks, only the information mapping needs to be changed, not the individual virtual object definition.

Collectors are used to define required combinational Boolean or mathematical functions for application programs requiring the combination of several input sources to drive a single output virtual object. Collectors

can also be used to determine a critical state by comparing an input value with a threshold value.

Data connections are also used to interface input objects of a prototype with an external application program. For example, interactive application programs may require control and display data interfaces to a prototype. The interface provides data communications between virtual objects and externally provided data. The prototyping development system also defines and provides communications with databases using connecting variables and remote or local communications.

The Logic Editor is used to model the prototype system by defining prototype system states and events. The system uses a Finite State Machine (FSM) concept as the framework for MMI decision-logic development. This allows the prototype system to manage events in a context that takes system history into account and supports a fine-grained modelling system functionality, thereby enhancing system reliability.

Prototype system behavioral modelling with the FSM concept produces a model containing a finite number of states. Each prototype state is entered and exited based on the occurrence of a virtual event. Events can be user or application generated.

To handle FSM modelling, the Logic Editor uses an Augmented Transition Network (ATN) language. The ATN language provides a structure for recognizing events and servicing transitions between prototype system states. Transitions are serviced by invoking routines and then transitioning or moving to the next state.

The Logic Editor is used to produce an ATN program and define prototype system operation. The ATN language has two basic constructs, the STATE and the EVENT. These constructs map to the nodes and links of a conventional state diagram to define prototype interface decisions, i.e., provide logic control and action routines. The general structure of an ATN program is as follows:

```
STATE start_state
{ initialize the state's parameters if required }
25 { EVENT event1 } (condition) { response }
    -> NEXT_STATE
    { EVENT event2 } (condition) { response }
30    -> NEXT_STATE
```

etc.

When a virtual event occurs, the prototype system is in a particular predefined state. The prototype system checks that the event that has occurred matches one of the transitions (i.e., an event and a condition) declared in that state. If the event is matched and the condition is true, i.e., valid and passed, the prototype system executes the response by executing a number of routines. The prototype system then proceeds to the next state, awaiting other virtual events. Thus, the ATN code is used to define decision making of the prototype system.

Input generated events can be received by the ATN program and output virtual objects can be addressed by action routines invoked by the ATN program. The MMI can respond to virtual and real events originating from many sources, including data entry, selection, discrete device input (such as a button, switch, input field, etc.), a continuous discrete device input reaching an increment (such as a potentiometer, knob or locator), time-out, periodic event, variable value crossing a threshold, voice input or a user-definable event.

In order to service events, action routines are dispatched, i.e., invoked to service the event. The prototype development system makes libraries of action routines available to fulfill a variety of functions including performing specific object operations, manipulating graphical images and virtual objects on a virtual CRT screen, entering line types, patterns, fonts, and colors under program control, performing geometric transformations under program control, writing to output fields and reading from input fields, type checking for strings, communicating with an external simulation via Ethernet or other specialized buses, and remapping object plugs.

Once the functionality of the user interface is completed using the Object Editor, Integration Editor, and Logic Editor, the dynamics of the prototype are displayed at runtime in the Runtime Environment and the user is allowed to interact with the prototype through various real and simulated input devices. The Runtime Environment permits a user to evaluate the prototype and operation of the final MMI.

Using such a prototype development system, an essential part in the programming the human-computer interface for the prototype is to specify the behavior of the interface to virtual events. Such events may be generated by input devices (buttons, locators and triggers, keyboards, selects, function keys, touch screens, pictorial touch panels, voice interfaces) as well as by a timeout, a message from another system or a variable crossing a threshold value.

It is convenient to use a Finite State Machine (FSM) paradigm to respond to virtual events. Use of a FSM is suited to addressing sequencing problems often found in operator interfaces, and allows structuring of the control logic in an orderly, regular fashion.

When an FSM is used, virtual events are processed in the context of a history of prior events and system states. A combination lock provides a good example of events interpreted in a context of prior events. Only the correct sequence will open the lock.

In another example, the pilot command "landing gear up" should only be serviced in the state "Weight-off-wheels" and not in the state "Weight-on-wheels". In a FSM, the system is deemed to be in one of a finite number of states. When input arrives, a transition is executed from the present state to the next state. The next state depends solely on the input and the present state. The output also depends solely on the input and the previous state. When applying the FSM model to operator interfaces, FSM input is made up of discrete events and FSM output is the dispatch of a series of subroutines or setting of plug variables. If a logical condition is tested before undertaking the transition, the FSM is an ATN.

ATNs or FSMs can be programmed with a text editor, by completing entries in a table or with a pictorial editor. However, textual completion of the table requires an in-depth knowledge of the syntax and structure of the ATN or FSM.

Accordingly, a need exists for an improved method and structure for defining the operation of a simulation system without requiring a knowledge of programming, FSM or ATN languages.

A further need exists for a method of, and structure for, defining a State Table using a graphical input and pull down, context sensitive menus.

A need further exists for a system to visually relate input and output virtual objects using a graphic input device.

A need further exists for a system which presents a user selection of valid relationships between virtual objects and available states.

A need further exists for a tool to specify a desired behavior using an graphic input device such as a mouse to point to, and designate, virtual objects which are part of the behavior and by designating desired virtual object behavior from menus.

### Summary of the Invention

The invention is directed to a context responsive menu driven prototype development and simulation system responsive to a graphic input device for defining relationships between and among prototype virtual objects displayed on a graphics display such as a video monitor. The relationships are defined by user completion of a State Table defining prototype system states, virtual events or triggers, virtual actions, and next states. Parameters are input to the table using a combination of images of objects appearing on the computer screen together with graphically designated menu selections and alphanumeric keyboard entries. Upon completion of the State Table, the parameters are translated into an Augmented Transition Network (ATN) or similar language for execution on a target platform running the simulation (i.e., workstation, personal computer, main frame, etc.). Thus, the prototype development and simulation system and method according to the invention directs the user to relevant valid choices defining relationships between virtual objects; displays and stores the relationships, actions, events, and responses as a spreadsheet-like State Table; and translates the State Table into a compilable program or executable language to be run on a target platform (e.g. computer) system.

According to the invention, prototype system performance is defined using a graphical input device to designate virtual objects and define relationships between and among the selected virtual objects. State Tables in the form of spreadsheets are completed by interacting with the image of an operator interface composed of virtual objects, and deriving virtual events or actions by clicking on the appropriate virtual objects.

Messages sent by a virtual object, or methods executed by it, can be arranged in a list (e.g., in a pull-down menu) and imported into the spreadsheet. For example, the event "Light XYZ is turned OFF" would be generated by clicking on the image of the object light XYZ, and extracting the event "Turn OFF" from a pop-up menus. Actions impacting virtual objects are specified in a similar manner.

In conjunction with the prior prototyping systems described, which provide a graphical design environment for virtual object appearance, virtual object behavior and virtual object interfaces, the invention closes the loop for allowing full pictorial programming in the domain of dynamic, graphical operator interfaces. Pictorial programming allows computer users who do not have a traditional programming background to carry out tasks that normally require programming knowledge, thereby expanding access to computer simulations.

The invention is further directed to a visual method of, and system for, programming interactive, graphical applications on a computer system which includes a graphical display and input devices, such as a mouse and a keyboard. The method is particularly well suited to programming graphical, dynamic, interactive operator

interfaces. Dynamic operator interfaces include applications with a graphical output which changes to reflect the values of data represented by virtual objects.

The visual application is made up of virtual objects, which have both a graphical representation and a behavioral model. The graphical representation of a virtual object is defined using a graphics editor, or otherwise imported into the prototype system. A behavioral model is specified by a Property Sheet that maps a behavioral model for a virtual object to the visual representation. The mapping is specified by pointing to features of the virtual object, such as line of motion or center of rotation, or typing-in values.

The behavioral model is used to animate the virtual object in a manner consistent with the class of object. For example, a "Dial" type of virtual object may be predefined to rotate while a "Scale" type virtual object translates, and a multi-state "Light" type virtual object assumes one of several graphical representations to simulate a physical light.

Virtual objects are connected to, or associated with, data values so as to animate the objects in response to the data. As the data changes, the virtual objects are redrawn to reflect the changed data value. This data can be generated by physical input devices, other virtual objects, software programs residing on the same or another computer, etc.

The subject computer prototype development and simulation system allows the user to specify the reaction to events in a pictorial manner, in a visual object environment. For this purpose, a spreadsheet like State Table and the visual object collection coincide on the same graphical display. The State Table is filled in by pointing to lists of virtual events or actions associated with the different virtual objects. The contents of these lists are dependent on the virtual object class. Event or action descriptions are entered into the respective cells of the State Table, in the form of descriptive strings of text. This text describes the events or actions, and the event sources, or action destinations.

The State Table allows the user to specify prototype state names, logical conditions, and the name of the next prototype state for processing the next virtual event, thus implementing an ATN control paradigm of the user interface. This information is supplied by a user employing an alphanumeric keyboard, selecting from lists, or copying parameters from other cells of the spreadsheet State Table. Other control paradigms can be implemented using the same method, such as FSM, Rule based systems, or Petri Nets. The data stored in the State Table is used to automatically derive the control program for the prototype.

Furthermore, in conjunction with the information stored in the virtual objects, a complete source code program can be generated. The source code program, when compiled, linked, and executed in the presence of appropriate data, produces the same visual appearance, animation behavior and interaction behavior to occur on a host computer system as on the original prototype development platform.

The invention includes a State Table Editor as an interactive module for specifying the behavior of a prototype using, for example, the commercially available Virtual Applications Prototyping System (VAPS). The State Table Editor or definition module interface is similar to a spreadsheet paradigm, in that the behavior of a prototype, i.e., responses of virtual objects to data, events, and other virtual objects, is defined by completing a two-dimensional table, using a modern point, click and type interface.

The State Table Editor replaces the Logic Editor of VAPS to define relationships between and among virtual objects, events, states and actions. The State Table Editor is compatible with various simulation and prototyping systems. Thus, Augmented Transition Network (ATN) language programs previously written using a Logic Editor are usable in the subject prototype development and simulation system. In addition, ATN programs can be converted into State Tables and State Tables can be converted into ATN programs.

The State Table Editor allows the user to specify the behavior of a simulator prototype using a spreadsheet like interface. This spreadsheet is used to specify the desired behavior in terms of behavioral rules. Each behavioral rule is called a "Reaction". A reaction specifies how the prototype should respond to a specified event. In its simplest form, a State Table specification for a prototype consists of a collection of Reaction Rules. This collection of rules is called a Reaction Table. More sophisticated prototypes can contain more than one rule set. Each rule set is called a state.

The State Table Editor has a "Point and Click" interface allowing the user to complete the State Table by clicking on the prototype elements, i.e., designating a virtual object by manipulation of a graphics cursor using a graphic input device such as a mouse. Then, using the mouse and/or a data entry device such as a keyboard, the user fills in the State Table to specify the desired behavior of the virtual object.

The State Table Editor package includes context sensitive menus and windows to enable an inexperienced user to define prototype behavior. It allows the expert user to describe even complex behavior, by providing access to an entire computer programming language such as "C" or ADA.

For example, if the user is specifying the behavior of a virtual object, then the State Table editor displays or "pops-up" a window which enumerates all the functions which can be performed by that virtual object.

If the virtual object is an input object, then all the input events which that virtual object can produce is

shown in a menu. If the virtual object is an output object, then all the functions which can control the virtual object is displayed. Once the appropriate event or function is chosen, a second pop-up window allows the user to fill in the details/parameters of the virtual object. Default values are nominally provided. If the parameter can have only a predefined set of allowable values, then these values are also displayed in a pop-up menu.

The State Table Editor converts the sequence of key clicks and typed values into the appropriate textual syntax. The textual syntax describing an event is the ATN syntax for finite state machines (FSM). Actions are described in a programming language such as "C".

The State Table Editor provides a front end to the ATN capability of the prototype development system. The State Table defined by the user is translated into an ATN program which is compiled and used to drive the target prototype system at runtime.

The State Table Editor allows a user who has little or no programming experience to specify the desired behavior of a prototype by using a mouse to point to the virtual objects which are part of the behavior, and by choosing the desired behavior from context sensitive menus.

The foregoing and other objects, features, aspects and advantages of the invention will become more apparent from the following detailed description of the present invention when taken in conjunction with the accompanying drawings.

#### Brief Description of Drawings

- Figure 1 is a block diagram of a processor platform according to the invention;
- Figure 2 is a blank state transition table;
- Figure 3 is a display used to define a reaction rule;
- Figure 4 is a display used to define a reaction table for an input-virtual object;
- Figure 5 is a State Table implementing the reactions rules specified in the State Table of figure 4;
- Figure 6 is a State Table defining the operation of the depicted virtual objects;
- Figure 7 is a chart of reaction trigger sources;
- Figure 8 is a State Table defining using a FIRST\_TIME trigger to define prototype responses;
- Figure 9 is a display depicting a State Table using an INITIALLY trigger to define prototype responses and a simulation virtual object;
- Figure 10 is a State Table including an IMMEDIATE event;
- Figure 11 is a prototype modified to include two virtual control knobs;
- Figure 12 is a flow chart depicting four phases of using the State Table Editor according to the invention;
- Figure 13 is a screen display according to the invention;
- Figure 14 depicts the display format of a Parameters Property Sheet;
- Figure 15 depicts the display format of another Parameters Property Sheet;
- Figure 16 depicts the display format of Parameter Value Windows;
- Figure 17 is a flow chart of State Table Cell selection according to the invention;
- Figures 18-20 are flow charts of State Table Cell configuration performed according to the invention;
- Figure 21 is a table of EVENTS for two example classes of objects.
- Figure 22 is a table of EVENTS not requiring specification of an object.
- Figure 23 is a table of ACTIONS for two example classes of objects.
- Figure 24 is a table of ACTIONS not requiring specification of an object.
- Figure 25 is a flow chart of system operation in a relevant choice mode of operation;
- Figure 26 is a flow chart depicting determination of default parameter values according to the invention;
- Figure 27 is a flow chart depicting system operation in response to a user parameter selection;
- Figure 28 is a flow chart depicting system operation in response to a user parameter value assignment selection;
- Figure 29 is a graphic image of a simulation display; and
- Figure 30 is a completed state transition table.

#### Best Mode for Carrying out the Invention

A computer platform for the development and simulation of a prototype is depicted in Figure 1. A central Processor 102 is connected to a Math Coprocessor 104 for executing arithmetic and logical operations responsive to a program of instructions stored in Random Access Memory 106. The program of instructions includes an operating system such as Unix, software components of the prototype development system including (i) an Object Editor, (ii) an Integration Editor, (iii) a State Table Editor, and (iv) a Runtime Environment module, and data generated by each to define the prototype. The Runtime Environment module also supports execu-

tion of a simulation of the prototype in response to prototype behavior specified by a State Table defined by the State Table Editor.

To speed memory access, the platform includes a Cache Memory Controller 108 and Cache Memory 110 for storing frequently used instructions and data. Mass system storage is provided by Disk Storage 112 and Disk Controller 114. Central Processor 102 can access Disk Storage 112 through Disk Controller 114 either via Input/Output Processor 116 or over Data/Instruction Bus 118 under the control of Bus Controller 120. Input/Output Processor 116 also receives and formats data from Keyboard 122 and Graphics Input Device 124 via Graphics Input Interface 128, and Touch Panel 128. Graphics Input Device 124 may include a mouse, joystick, light pen, digitizing tablet, trackball, keypad, or other device for positioning a video cursor.

The resultant prototype simulation is generated by Graphics Processor 130 under the control of Central Processor 102. Graphics Processor 130 generates alphanumeric and primitive graphic image data either under direct control of Central Processor 102 or in response to lists of instructions or vectors stored in Random Access Memory 106. The image data is stored as a bit mapped image in Image Memory 132 and displayed by Video Monitor 134.

The computer platform also includes an External System Interface 136 for exchanging data with another computer system. Speech recognition is supported by Speech Recognition Processor 138 which receives speech sounds at microphone 140, interprets the sounds and provides a corresponding ASCII output string to Input/Output Processor 116. Finally, synchronous type operations of the computer platform are synchronized by System Clock 142 which also provides periodic interrupts signals at predetermined intervals set under software control.

Referring to Figure 2, the State Table is a two dimensional table with four columns and an unlimited number of rows. The four columns are:

- 1 - State
- 2 - Event or Trigger
- 3 - Action
- 4 - Next State

A State Table represents the behavior of a prototype simulation as a collection of Reaction Tables. The present invention provides a method of, and apparatus for, filling in the state table of Figure 2 by controlling virtual objects on Video Monitor 134 in response to user input signals derived from any or all of Keyboard 122, Graphic Input Device 124 or Touch Panel 128. Context driven windows are displayed on Video Monitor 134 in response to user designation of cells of the state table to be completed, the windows providing lists of valid table entries for the selected cells.

A reaction rule specifies how the prototype should respond to events. The Prototype must respond to a user operating real input devices in the which affect virtual input objects of the prototype. Using Keyboard 122, Graphics Input Device 124 and Touch Panel 128, the user may push a virtual button, rotate a virtual knob or slide a virtual potentiometer graphically represented on Video Monitor 134. The simulator system responds to the inputs by affecting the output virtual objects of the prototype as displayed on Video Monitor 134.

For example, referring to Fig. 3, in response to the user activating a joystick or mouse of Graphic Input Device 124 or by manipulating the corresponding object using a touch screen to cause simulated rotation of the illustrated virtual knob the simulation system adjusts the illustrated virtual speed dial to display the value of the knob. This is described in a reaction rule as follows.

"KNOB speed\_knob"

which is called the event part of the reaction rule, while

"Drive\_Dial(" ", "speed\_dial", KNOB\_VAL);"

is called the action part of the reaction rule. The event or trigger part of the reaction rule is written using a syntax which is unique to the simulation system (i.e., ATN), while the action part is written in the "C" programming language.

More complex reaction rules are possible. For example, referring to Figure 4, the prototype may contain a virtual knob and a virtual light selectively displaying one of two illumination levels, e.g., black and white. In the example, the light turns white whenever the value of the knob is less than 500, and turns black when the value of the knob is greater than or equal to 500.

This relationship can be expressed with a pair of reaction rules. A collection of one or more reaction rules is called a **reaction table** as depicted in Figure 5. The reaction table depicted in the figure has two reaction rules, with the event part of both reaction rules being two lines long. The first line gives the primary event or trigger which must occur for this rule to be triggered. This **primary event** is the knob turning. The second line of this reaction rule is called the **condition** of this event. The condition specifies that this reaction rule should only be taken if the value of the knob is < 500. Conditions are also written in the "C" programming language.

Each of the two reaction rules have the same primary event as triggers. The rules differ only in the con-



dition which must be true for the reaction to take place.

In the prototype development system, a user enters a reaction table in the form of a state table. A prototype development State Table consists of one or more reaction tables. Only one reaction table is active at any given time. A simple simulator system State Table consisting of only one reaction table is depicted in Figure 5.

Since the State Table has only one reaction table, the State and Next State columns of the State Table are left blank. When a State Table has more than one reaction table in it, each reaction table must be given a name. The name of each reaction table is called a state. When a State Table is executing, only one reaction table within the State Table is active at any given time.

A State Table thus consists of one or more states. When a prototype whose behavior is specified using a State Table is running, the prototype simulation system will be in one of the states given in the State Table. Thus, the word state is synonymous with "mode".

A prototype may be controlled by more than one State Table. For example, different virtual objects may be controlled by separate State Tables.

Each reaction rule can specify that a change in state should occur as part of the reaction to the rule being triggered. This change in state occurs in response to an Event and satisfaction of the corresponding Condition. When the event occurs, the prototype executes any actions specified in the action column in the rule. After the action has executed, the state of the State Table changes. Once the state of the State Table has changed, only the reaction rules specified for that new state are active.

Figure 6 depicts a simple prototype and State Table having two states. The two states are called LOW and HIGH. When the State Table is in the LOW state, the value of the knob is copied to the dial. When the State Table is in the HIGH state, the value of the knob is doubled and copied into the dial. Flipping the GEAR switch toggles the State Table from the LOW to the HIGH state and back again.

Each of the two reaction tables shown in Figure 6 has two reaction rules. The first reaction rule in the LOW state specifies an action in response to the user turning the knob. Since turning the knob does not change the basic mode of the Prototype, no mode change is required. Therefore, the NEXT STATE column is left empty.

The second reaction rule is also part of the LOW state. It specifies that whenever the switch is set to HIGH the State Table switches to the HIGH state. The dial is also reset to twice the value of the knob.

The HIGH state is structured similarly to the LOW state, except that, in the HIGH state, twice the value of the knob is sent to, and displayed on, the dial. As also defined in the HIGH state, the prototype simulation reverts back to the LOW state when the switch is switched back to LOW.

Another term used to refer to reaction rules is transitions.

The reaction rules described above all had events which were triggered by Virtual Input Objects. Other reaction triggers are possible. Reaction triggers can have one of the following sources:

- 1 - An input virtual object generating events;
- 2 - Clicking the mouse on the screen in an area unoccupied by a virtual object;
- 3 - Events generated by Computer Programs;
- 4 - Input from a Serial device or communications line;
- 5 - Time based events;
- 6 - Variables crossing thresholds; and
- 7 - Inputs from real devices which are mapped to virtual objects.

Other reaction triggers are shown in the table of Figure 7. The first group of events are examples of object events, i.e. events which are generated by using the mouse with virtual objects in the prototype. Some virtual objects have more than one input event associated therewith. Locators (e.g., mouse or joystick) have three different input events. The LOCATOR\_DOWN and the LOCATOR\_RELEASE event occur upon pressing and releasing the mouse in the active area of the Locator. The LOCATOR\_DRAG event occurs when the location of the Locator is changed using the graphic input device on the active "touch" area of the display screen within the defined area of motion for the Locator.

Field events occur when the user types data into an input field, for which a "Read\_Field()" function has been invoked. A FIELD\_VALID event occurs if the value typed in is valid with respect to the syntax for the field. A FIELD\_INVALID event occurs if the value entered is not valid. In either situation, a FIELD event (without the VALID or INVALID qualification) is generated.

Referring to the Other Mouse Input Events entry, a NODEVICE event occurs if the user clicks somewhere in the prototype that is not within an active area of a virtual object. This event is received only if it is enabled in the Runtime Environment using the "Enable No Device Event" field in the Operational Settings of the Configuration menu.

The ANYEVENT trigger, responds to any event that occurs from input virtual objects, as well as any PERIODIC event which occurs. This is useful to detect and log stray events.

The next five events are generated internally within FSM in response to state transitions. The **FIRST\_TIME** event occurs the first time that a given state is entered. This event does not occur a second time if the state is entered and exited and entered again. The **FIRST\_TIME** event is often used to perform initial loading of a frame, wherein a frame is a collection of objects.

The **INITIALLY** event occurs each time that the a new state is entered from another state. This event is used to configure the display for the newly activated state. This may include the loading frames or changing the values of variables used by other transitions within that state. In contrast, the **FIRST\_TIME** event is used to load the frame with the virtual objects only the first time the State Table is initially entered. Thus, the next transition has an **INITIALLY** trigger. In the previous example, whenever this transition is entered from another state, the dial is reset to the value of the knob. The dial is also set to the value of the knob each time a new input value is received from the knob. When the user toggles the gear switch to **HIGH**, the mode is switched to **HIGH**. There is no action associated with this state. Instead, the system has only associated a change in state with this reaction rule. Once in the **HIGH** mode, the **INITIALLY** reaction is performed. This causes the dial to be set to twice the value of the knob.

The use of the **INITIALLY** event helps make the State Table clearer and easier to understand. Compare the State Table shown in Figure 8 with that shown in Figure 9. In the State Table of Figure 8, it was necessary to place an action in the transition to cause the state to change when switching from one mode to another. Although this resulted in the desired behavior, it is conceptually incorrect, since all the uses of the action routine to set the value of the dial to twice the value of the knob should be encapsulated within the **HIGH** state.

The **IMMEDIATE** event is invoked each time that a state is entered, as well as after each reaction in which the state did not change. An application of the **IMMEDIATE** event is to perform data logging.

Referring to Figure 10, different transitions are taken in response to different input events. In all cases of a transition being taken, a log entry is made. Another use of the **IMMEDIATE** event is when there is an output virtual object in the display which can be affected by one of many input virtual objects. In these cases, rather than driving the output virtual object from within, transition routines can be factored out which drive the output virtual object, and place it in the **IMMEDIATE** action. An example of such a use of this event is depicted in Figure 11.

Referring to Figure 11, the prototype includes two knobs. The prototype displays the sum of the values represented by the two kinds on the dial. This is accomplished by storing the value of the knob in the reactions which respond to the knobs being turned, and driving the dial with the sum of the values in the **IMMEDIATE** event response.

The **FINAL** event occurs when leaving a state. It occurs only if there is a **NEXT\_STATE** filled in, and the **NEXT\_STATE** is different from the current state. In this case, the action associated with the **FINAL** event is executed before the transition to the next state. A typical use of the **FINAL** state is to undo any actions performed in response to the **INITIALLY** event upon entering the state. A typical example is unloading of frames which were relevant only within that state being left.

The Pseudo Event **COMMENT**, has no operational effect. The remainder of the line is treated as a comment and ignored. Comments can also be entered as "C" language comments, i.e. between **/\*** and **\*/**. For example:

```
/* This is a comment */
```

The prototype design system also provides two time based events. The first one is called **PERIODIC**. The **PERIODIC** event occurs repeatedly at the specified frequency. The syntax for this is

```
PERIODIC nn HZ
```

where **nn** is the frequency of the event in HERTZ. The **PERIODIC** event can be used to increment counters, or implement samplers which change the display at specified intervals.

The clock which generates the **PERIODIC** event is a global clock. This means that the **PERIODIC** clock is not reinitialized upon entering a state.

The other time based event is the **ON TIMEOUT** event. The syntax for this is:

```
ON TIMEOUT nn SECS
```

In this notation, **nn** must be a positive integer. This event occurs **nn** seconds after entry of the state and is used, for example, to detect if the user has failed to timely enter data or to determine if the user has finished entering data. The **ON\_TIMEOUT** is reset upon occurrence of a reaction or state change.

In addition, there are two user defined events. The first user defined event is called **USER** which is generated by a user program. The **USER** event takes two parameters; the name of the **USER** event and the value of the event.

The syntax for using **USER** generated events is:

```
USER User_Event_Name User_Event_Value.
```

The **USER** event is generated within another ATN using the routine **Generate\_User\_Event**. The syntax is:

Generate\_User\_Event("User\_Event\_Name", "User\_Event\_Value");

User generated events are used where multiple State Tables are concurrently executing. These State Tables can signal each other using customized events.

As described above, the prototype development system consists of four main components. The Object Editor (OE) allows the user to draw virtual objects by specifying the shape, color, placement and texture of the graphical objects and ascribe to each an appropriate class of behavior. For example, if a dial is drawn, it can have any shape, or color.

The Integration Editor (IE) allows the user to specify the names of data variables which control the shape of virtual objects. For example, if a virtual object is a dial, a particular variable can be made to control the angle of the "needle/pointer" on the dial.

The third component is the State Table Editor. This component allows the user to specify more complex behavior than possible using the Integration Editor of the previous systems. For example, in the Logic Editor (LE), the user can arrange to have virtual objects appear or disappear when a particular button is pressed and a particular condition is outstanding. The State Table Editor instead uses State Transition Programs, augmented with "C" programming code. The State Table Editor replaces the LE of previous simulator systems.

The fundamental difference between the State Table Editor and the LE is ease of use. The State Table Editor significantly reduces the user generated and supplied programming code required to specify the behavioral specification for a simulator prototype.

The final component of the simulator is the Runtime Environment (RE). The RE integrates the information entered using OE, IE and State Table Editor and runs the specified interactive graphical display.

There are four phases to using the State Table Editor as shown in the flow chart of Figure 1. When invoking the State Table Editor, the user initially sets up the environment of the prototype development system. Once the user has set up the environment, the State Table Editor can be used. The prototype development system records how the user set up the environment the last time the State Table Editor was used.

The following is a description of setting up the environment and editing the State Table.

The State Table Editor consists of the following components:

1. The State Table Spreadsheet;
2. Displayed Collection of Graphical Objects called a Frame;
3. Command Menu Bar
4. Text Entry Window; and
5. A Displayed List of Choices that are relevant to the current mode.

A typical State Table Editor environment is shown in Figure 13. The State Table includes a menu bar 30 describing options available in the State Table Editor. A Frame Display 32 graphically depicts virtual objects to be defined by the State Table Editor. Relevant Choices List 34 provides user selectable commands. Text Entry Window 36 allows the user to enter virtual object labels and ATN instructions. The resultant system definitions are displayed in State Table 38.

By editing a State Table, a user specifies the behavior of a particular graphical display, i.e., a Simulator Picture Frame. This is a frame created using the OE. This frame contains one or more graphic objects. The user selects one such frame to be controlled by the State Table using the State Table Editor.

The user also designates whether a new State Table is to be created or if an existing State Table is to be edited. In the latter case, an existing State Table is retrieved and loaded into memory for editing. During the editing, the user may load a different State Table, or a different frame. The other components of the environment are placed in the environment by the State Table Editor without user intervention.

Loading a frame or a preexisting State Table is accomplished by choosing the appropriate entry from the pull down menus in the menu bar, and choosing the desired file name.

Once the user has set up the State Table environment, the State Table can be edited. Editing the State Table includes filling in the cells of the State Table. Referring to Figure 13, the State Table is organized like a four column spreadsheet. The State Table can have as many rows as required to specify the desired behavior. Each cell in the State Table contains information specific to that cell.

For example, if a particular cell in the event column has as its "subject" a particular knob, then the relevant list for that cell includes the various events that can be generated by the knob. This may include the events "MOUSE\_UP", "MOUSE\_DOWN", "KNOB\_NEW\_VALUE", "KNOB\_HAS\_MIN\_VAL", "KNOB\_HAS\_MAX\_VAL". The cells in the state column and the next state column contain the names of states as previously designated by the user.

The relevant list for the state column is the list of all the states in the State Table. This includes any state which is already defined by another cell in the state or next state column.

The relevant list for the next state column includes all the states entered in the state table. In addition, certain keywords are also listed including "EXIT", "SUSPEND", "RESUME" and "RETURN".

Both the cells in the Event column and the Action column can refer to a virtual object as their "subject". Relating a cell to an object is performed by clicking on an event or action cell, and then clicking on a virtual object in the frame. The relevant list for a cell in the Event column depends on the virtual object referred to in the cell.

If no virtual object is associated with a cell, then a list of events that do not require a virtual object is displayed. This includes events such as "STATE\_TABLE\_STARTED", "STATE\_ENTERED", and "STATE\_EXITED". If a virtual object is associated with the cell, the relevant list then includes events that can be generated by the virtual object. For example, if a particular cell in the event column has an associated virtual object comprising knob, then the relevant list for that cell includes the various events that can be generated by the knob. This may include the events "MOUSE\_UP", "MOUSE\_DOWN", "KNOB\_NEW\_VALUE", "KNOB\_HAS\_MIN\_VAL", "KNOB\_HAS\_MAX\_VAL".

The relevant list for a cell in the Action column depends on the virtual object which is referred to in the cell. If no virtual object is associated with the cell, then actions that do not require a virtual object are included in the relevant list. This includes actions such as "CHANGE\_STATE\_TABLE\_PRIORITY", "START\_COMMUNICATIONS", and "TERMINATE\_PROGRAM".

If a virtual object is associated with the cell, then the relevant list comprises a list of actions which can be performed on the virtual object. For example, if a particular cell in the action column has as its associated virtual object a particular multi-state light virtual object, then the relevant list for that cell includes various actions that can be performed on the specified virtual object type. This may include the actions "CHANGE\_STATE", "CHANGE\_TO\_DEFAULT\_STATE", "HIDE", "REVEAL", "BLINK".

The list of events and actions which are associated with each object type is defined by a user alterable dictionary including a list of events and actions supported by each object type. This allows the set of virtual object classes to grow, or the list of events and actions which virtual objects support to change, without changing the state table program source code.

Once an event or action is chosen, the event may be assigned one or more parameters. If a parameter is to be assigned, an additional window pops up and to allow the user to enter values for the parameters. Typical parameters property sheets are depicted in Figures 14 and 15.

Each field in the property sheet has a "set of allowed values". Some fields may receive numeric values, while others receive a name of a state of the virtual object, and others require the user to choose from a particular list of values. When the user clicks on a particular field in the property sheet, a parameter value window pops up, i.e., is displayed. This parameter value window allows the user to enter the value for the particular parameter. If the parameter has a fixed set of allowed values, a list of those values is shown. The user then clicks on the desired value.

The list of allowed values may come from one of two sources. It may come from the virtual object which is the subject of the cell being edited. For example, if the virtual object is a multi-state light, the names of the states of the light may be shown. Alternatively, the field may be defined as a "C" or other computer language type. If the field type is an enumerated type, i.e., limited to a predefined set of values, the list of allowed values is taken from the enumerated list for that type. For example, if the type of the field is ANCHOR\_TYPE and the type was defined as:

```
typedef enum
{RELATIVE_TO_CRT, RELATIVE_TO_FRAME, ABSOLUTE,
  SCALED_TO_FRAME}
ANCHOR_TYPE;
```

then the appropriate values are used for that field.

Finally, if the field is numeric or an arbitrary string, then a text entry field is used. Examples of parameter value windows are depicted in Figure 16.

The basic sequence of completing or filling in a State Table is to select a cell by clicking on it and either typing in a value using the edit text window or clicking on the appropriate items from other windows and menus. This process is depicted in the flow diagram of Figure 17.

Initially, the user selects a cell and the system configures the State Table Editor for the selected cell. Details of the system table configuration routine are shown in Figures 18-20.

Referring to Figure 18, the State Table Editor first removes any preexisting windows from the display. If the selected cell is a State cell, then a list of state cell keywords and all known state names are displayed in a window. If the cell is a Next State cell, then a list of next state cell keywords and all known state names is displayed in a window area. If the cell is an Event cell, event cell processing is performed in accordance with

the flow diagram of Figure 19.

Referring to Figure 19, if the cell has a subject, then the relevant choice list displayed includes a list of events for that virtual object type and the specified cell subject. If there is no subject associated with the Event cell, then a list of event corresponding to that type of cell is displayed. A timeout is an example of an event not requiring specification of a subject.

The list of relevant choices depends on the class of the object, the list being different for different object classes. Examples of lists of events related to specific object classes are depicted in Figure 21. One such list is needed for each object class. Similar classes of objects may have some events of each list in common. The lists are stored in a data base.

If the cell does not have a subject, then a list of events which require no subject is displayed. An example of such a list of events is shown in Figure 22.

After the relevant choices associated with the Event cell are displayed, a check is performed to determine if a relevant choice for the cell has already been selected. If so, the current parameter values are set accordingly and displayed in an Event parameters window.

If the cell is not an Event cell, then the cell is interpreted as an Action cell. Referring to Figure 22, a relevant choice window is displayed corresponding to whether a subject is associated with the Action cell. Action cells not requiring a subject include routines contained in a "C" library such as for computing a random number. The system further displays a Relevant Action parameters window if a cell has a relevant choice already chosen.

The list of relevant choice actions is determined in a way similar to that used to determine relevant choices of events. Example of lists of relevant actions for different object classes are depicted in Figure 23.

If an action cell does not have a subject, then a list of actions which require no subject is displayed, as shown by example in Figure 24.

Referring again to Figure 17, once configured, the system waits for an input from a user. The input is classified into six types: the user (i) types text into an edit window, (ii) picks an edit command from a window, (iii) chooses a virtual object from a frame, (iv) selects an item from a relevant choice window, (v) selects a parameter from a window, or (vi) selects a parameter value from a parameter value window. If the user enters text, the system waits until the user finishes typing and supplies the text to the prototype development system for parsing and further processing.

If the user designates an Edit command, the State Table Editor performs the chosen editing function. Selection of a virtual object causes the State Table Editor to make the selected virtual object the subject of the cell.

If the user selects a relevant choice from a window, processing proceeds as shown in the flow chart of Figure 25. As depicted, the selected choice is assigned to the cell. If the choice requires one or more parameters, processing is performed as shown in the flow chart of Figure 26 to determine an appropriate default value for each required parameter.

Referring to Figure 26, a check is first performed to determine if the relevant choice provides a default value. If not, then, if the type of choice is a characteristic of the virtual object which is the subject of the current cell, then a default parameter is provided corresponding to the specification of the virtual object. Alternatively, if the relevant choice is not a characteristic of the virtual object, then a default value is supplied based on the type of choice selected.

Referring again to Figure 17, if the event or action window is up and the user selects a parameter from the parameter window, processing continues as shown in Figure 27 to allow the user to select a parameter to be specified. The chosen parameter is then made the current value. Alternatively, if the parameter value list window is up and the user selects a parameter, processing continues to allow the user to choose a parameter value as depicted in Figure 28.

The following description is an example of using the State Table Editor and prototype development system according to the invention.

The user first enters the Object Editor by typing 'object\_editor' from the command prompt or by clicking on a command icon at a computer workstation. Then, the user defines or retrieves a previously stored prototype. In this example, the prototype is the simplified control panel 20 for a train as shown in Figure 29.

As described and defined using the Object Editor and Integration Editor, the prototype has two output virtual objects and two input virtual objects. The two output virtual objects are traffic light 22 and speedometer 24. The input virtual objects are push button 26 and a gas pedal represented by potentiometer 28.

For example, suppose that the operator determines that the control panel is to exhibit the following behavior. The main control is to be the speed control, i.e., sliding potentiometer 28. The train can travel at any speed from 0 to 200 miles per hour. Control panel 20 includes traffic light 22 which informs the operator of the maximum allowed train speed. If traffic light 22 is green, then the train may proceed at any speed. However, as indicated by the crosshatched area on the speedometer, speeds greater than 180 mph are dangerous and

should not be used except under special circumstances.

When traffic light 22 is yellow, the train may not travel faster than 20 mph. When traffic light 22 is red, the train should stop.

- 5 In the prototype, traffic light 22 should be controlled by push button 28. Repeated pressing of push button 26 causes the traffic light sequence through the colors red, green and yellow. Initially the traffic light is red. Pressing button 26 once makes it turn to green. Pressing it again makes the light change to yellow. Pressing it again makes the light change to red.

- After defining the desired operating rules, the user creates a State Table to implement these rules. The user creates an empty State Table using a main pulldown menu to select "State Table." This causes a display of a pulldown menu from which the user selects the "New State Table."

- 10 Next, the user brings up the Pick By Name window. From the main pulldown menu, the user selects "Utilities" and, from the resultant pulldown menu, selects "Pick By Name". Then, from the State Table definition main pulldown menu, the user clicks on "Utilities". From the Utilities menu, the user clicks on "Focus Selection". From Focus Selection, the user clicks on the second entry in the menu, connecting the Pick By Name menu to the frame which the user has loaded into a Show window for display. Then, the Focus Selection window is removed by clicking on an appropriate field of the Focus Selection window. The user can then begin to enter the desired prototype behavior into the State Table.

- The user must first arrange to have the Frame loaded when the prototype begins operation. This is accomplished by clicking on the first cell in the column of the State Table Editor labelled EVENT. In response, a window titled "Relevant Choices" is displayed. From this window the user selects the entry "FIRST\_TIME". The selected cell in the State Table Editor now contains the text FIRST\_TIME.

- The user next clicks on the first cell in the column labelled Action. In response, the "Relevant Choices" window changes to display context appropriate choices. In the "Pick By Name" Window, the user clicks on the top element, which is the frame itself. The "Relevant Choices" window now displays new choices applicable to the frame.

- 25 The user next selects the entry "Get\_Frame" from the Pick By Name window. As a result, a window titled: "Arguments of Function: Get\_Frame()" appears. This window allows the user to change the parameters to the Get\_Frame() action. Since the default parameters are appropriate, no change is required.

- The first state to be created is titled RED. The user initially clicks on the first cell in the column labelled 30 Next State, highlighting the display of the cell. As before, a Relevant Choices window will appear. The user clicks on the cell again causing a data entry window to appear. This follows the general rule in the State Table Editor that a second click on a cell causes a data entry window to be displayed, allowing the user to manually input values using alphanumeric keyboard entry.

- The user then types "RED" into the table followed by a carriage return from the keyboard. The value RED then appears in the NEXT STATE Cell. Above the State Table editor work area are State Table editing commands. The user clicks on the command "Append Transition" to add a second empty row to the State Table.

- Next, the user clicks on the second cell in the column labelled State. From the Relevant Choices menu, the user selects "RED" to create the state RED.

- In the state RED, the user first sets the dial representing the speedometer to 0. To do this, the user clicks 40 on the second cell in the EVENT column. From the relevant choices menu, the user selects the INITIALLY item. The initial behavior of the system in response to entry into the RED state is then specified in the Action column.

- To specify the behavior, the user clicks on the second cell in the ACTION column. From the Pick\_By\_Name window, the user clicks on and highlights DIAL. This causes the dial to be made the subject of the cell.

- 45 From the Relevant Choices menu, the user next clicks on the Drive\_Dial entry whereby windowArguments To Function: Drive\_Dial are displayed. Since all the default values are appropriate, no change is made.

- The user again clicks on the Action cell causing a data entry window to be displayed. The user adds a second blank line and selects the traffic light from the Pick By Name window. From the relevant choices menu, the user selects "Drive\_Light". Since the default value given is the RED mode of the light, no change is required.

- 50 The user must next describe the behavior while in the RED State. The behavior in the RED state is to go to the GREEN state when push button 26 is operated or "clicked". No response is required when the user changes the speed using the potentiometer in the RED state.

- To describe the behavior that should occur in response to button 26 being clicked in the RED state, another transition must be added. The user first clicks on the command "Append Transition". Since this transition is a continuation of the description of the behavior of the RED state, the user need not fill in the STATE name column.

- 65 The user then clicks on the EVENT column of the new transition which is the third cell in the column. From the Pick By Name window, the user clicks on the "Desired Speed Button." Alternatively, the user can click on

the picture of the button, using the left button on the mouse.

Next, from the Relevant Choices menu, the user picks the **BUTTON** selection resulting in display of default event parameters. Since, in the example, all the default values are correct, no change is required.

Having defined the **RED** state, the user next defines the **GREEN** state. To do this, the user clicks on the **NEXT\_STATE** cell of the transition. Clicking on the cell a second time causes a data entry window to be displayed. The user defines the name of the state by typing "**GREEN**".

Next, the user fills in the **GREEN** State parameters to define the behavior for the **GREEN** state. First, the user creates the **GREEN** state by clicking on the "**Append Transition**" command. Since, three transitions are needed in the **GREEN** state, the user clicks on the "**Append Transition**" command two more times, for a total of three times.

The user next clicks on the **STATE** cell of the first of the three transitions that have just been added. From the Relevant Choices Menu, the user clicks on **GREEN**. Now, the user clicks on the **EVENT** cell of that transition. From the Relevant Choices menu, the user defines the initial state by clicking on **INITIALLY**.

The initial behavior in the **GREEN** state is almost identical to the required behavior for the **INITIALLY** event of the **RED** state. Therefore, the quickest way to enter this action is to copy it from the **RED** state, and then change the details. To do this, the user first clicks on the action cell of the **INITIALLY** transition in the **RED** state. The user then clicks on the **COPY CELL** command. Next, the user clicks on the action cell of the first transition of the state and then clicks on the **PASTE CELL** command.

Next, the user clicks on the first row of the action cell causing a **Function Parameters: Drive\_Dial()** window to be displayed. The user clicks on the second field in this window. From the new pop-up window displayed, the user selects the **POTEN\_VAL** entry.

The user now clicks on the second row of this cell. The **Function Parameters: Drive\_Light()** window is displayed in response. The user clicks on the second field in this window. This brings up a window with a list of modes relevant to the light. In the present example, the light can be in the "**RED**", "**GREEN**" or "**YELLOW**" modes, these modes therefore being displayed in the window. The clicks on the "**GREEN**" entry thereby entering "**GREEN**" value as the **Drive\_Light** parameter. Alternatively, the user can click twice on the second field. In this case, a data entry window will be displayed into which the user can type the value "**GREEN**".

The next task in defining the operation of the prototype is to define the correct response to the user clicking on the button, or sliding the potentiometer. To do this, the user clicks on the second cell in the **EVENT** column of the transition. From the Pick By Name window, the user clicks on the potentiometer **Desired Speed**. Now, from the Relevant Choices menu, the user clicks on **POTEN**. Next, the user fills in the appropriate response to a change in value of the potentiometer.

To define the appropriate response to adjustment of the potentiometer, the user clicks on the **Action** cell of the transition. From the Pick By Name window, the user clicks on **Traffic\_Light**. From the relevant choices window the user clicks on **Drive\_Dial**. The **Function Parameters: Drive\_Light()** window is then displayed. In this window the user clicks on the second cell which has displayed therein **ZERO\_VAL**. A pop-up window displays available values. The user must then click on the **POTEN\_VAL** value.

To define the **YELLOW** State, the user first creates three transitions using the **Append Transition** command. The first transition describes the initial behavior of this state. The user first clicks on the **STATE** column of the first transition of the **YELLOW** state and **YELLOW** is selected from the relevant choices column. The user next clicks on the **EVENT** column, and selects **INITIALLY** from the Relevant Choices menu.

The desired behavior is to limit the speed to no more than 20 mph in the **YELLOW** state. To do this, the user types in a short program which implements the desired behavior. The program can be entered by clicking twice on the **Action** cell of this transition, causing display of a data entry window.

The program is entered into the cell as follows:

```
Drive_Light ("Traffic_Light", "Yellow");
float desired_speed;
desired_speed = POTEN_VAL;
if (desired_speed > 20.0) desired_speed = 20.0;
Drive_Dial("Speedometer", desired_speed);
```

Next, the user defines the response to the operator attempting to change the speed of the train. The user clicks on the **EVENT** cell of the second transition of the **YELLOW** state and selects the **Pick-By-Name** window select the **Speed** control.

The desired action for this cell is the same as for the INITIALLY action. To copy the action, the user clicks on the ACTION cell of the INITIALLY transition of the state, clicks on the command COPY CELL, and then clicks on the empty action cell below. To fill in the empty cell, the user clicks on the PASTE CELL command, causing the code to be pasted into the Action cell.

Finally, the user must describe the behavior if the user clicks on the change\_light button while in the YEL-LOW state. To do this, the user clicks on the last cell in the EVENT column. From the Pick-By-Name window, the user clicks on the Change\_Light window. From the relevant choices window, the user then clicks on the BUTTON event. Next, the user clicks on the last cell in the NEXT\_STATE column. From the relevant choices menu, the user selects RED by clicking on the displayed term.

This completes filling in the State Table, shown in completed form in Figure 30.

The user must next save and compile the State Table. First, the user clicks on the State Table Editor main pulldown menu. From this menu, the user selects the Logic Prototyping entry. This causes display of a flow chart like Logic Prototyping Control Panel.

In the Logic Prototyping window, the user should click on the command SAVE. This instructs the State Table Editor to save the State Table just entered into the warehouse. The user must give the State Table Editor a name by typing in "TRAIN", into the window which comes up, followed by pressing the <enter> key. The State Table is then saved.

To compile the State Table, the user must click on the COMPILE ATN command in the Logic Prototyping window. Finally, when the MAKE TASKS command in the Logic Prototyping window is highlighted, the user should click on the highlighted command. This completes the behavioral part of the prototype.

The final step is to run the State Table just created. To do this, the user invokes the Runtime component of the prototype development system. This is done by selecting the Session command in the main State Table Editor pulldown window. Next, the user clicks on the command Run Time Environment, initiating the Run Time components of the prototype development system.

Using standard simulation system procedures, the user selects the warehouse where the compiled State Table is located. From the Storage Contents window, the user first selects the file TRAIN, with a file type of ATN, loads, and runs the file causing the prototype to begin execution. The user operates and interacts with the prototype by, for example, changing the values of the speed control and pressing the button while watching the results on the speedometer and the traffic light.

As demonstrated, the process of creating a State Table is relatively simple. Most of the required information in a State Table can be entered with mouse clicks. When the default parameters that are entered into the State Table are not correct, the entries can be easily changed. When necessary, the user can click twice on any field, and manually type in or edit the desired text.

In summary, the prototype development system according to the invention provides a system and method for defining the responses and states of a simulated system using a series of context sensitive menus to generate state tables. The information from the state tables is used to generate program code used by the prototype development platform or a target platform to simulate a desired system.

Although the present invention has been described and illustrated in detail, it is clearly understood that the same is by way of illustration and example only and is not to be taken by way of limitation, the spirit and scope of the present invention being limited only by the terms of the appended claims. For example, although the State Table Editor has been described in the context of a system for simulating a physical system, the invention is also applicable to other systems such as real-time control, artificial intelligence, and other data processing systems the operation of which can be defined by a state transition table.

#### Claims

1. A real-time finite state prototype development and simulator system for graphically responding to user inputs by displaying virtual objects forming a modeled system on a monitor, comprising:
  - a graphic input device responsive to said user inputs for deriving a positional signal; and
  - a processor unit including:
    - (i) an input module responsive to said positional signal and to a position of one of the virtual objects for deriving selection data,
    - (ii) an object editor module responsive to said selection data for defining ones of said virtual objects as input and output virtual objects and ascribing to each a class of behavior,
    - (iii) a state table editor module responsive to said selection data and to said class of behavior ascribed to said virtual objects for defining a table of parameters describing the modeled system including:
      - (a) system states,



- (b) events to be monitored,
  - (c) actions to be taken in response to the monitored events, and
  - (d) next states to be entered by said modeled system in response to the monitored events, and
  - (iv) an execution module for detecting events and, responsive to said table of parameters and to said detected events, for selectively displaying and manipulating said virtual objects on the monitor.
2. The system according to claim 1 wherein said table editor module includes an interpreter module responsive to said table of parameters for supplying command data in the form of an Augmented Transition Network (ATN) Language.
3. The system according to claim 1 wherein said state table editor module includes a collection of reaction rules forming a reaction table stored in memory.
4. A finite state machine for simulating a system and responsive to a signal representing a present state of the simulated system and to a plurality of input signals for deriving an output signal including a signal defining a next state of the simulated system, said finite state machine including:
- a random access memory responsive to an address signal for selectively retrieving instruction signals stored therein; and
  - a processing unit for storing the present simulated system state and responsive to said simulated system state, the plurality of input signals and said retrieved instruction signal for supplying said address signal and defining the next state of the simulated system, said processing unit including:
    - (i) a state table editor module responsive to a user input signal for deriving a state table defining
      - (a) present states of the simulated system,
      - (b) input signals to be monitored by the simulated system,
      - (c) actions to be taken by the simulated system in response to the monitored signals, and
      - (iv) next states of the simulated system to be entered in response to the monitored signals, and
      - (ii) a runtime environment module responsive to said state table for loading said instruction signals into said random access memory.
5. A method of real-time finite state simulation for graphically responding to user inputs by displaying virtual objects on a monitor, comprising the steps of:
- deriving a positional signal representing a position of a cursor on the monitor in response to the user inputs;
  - supplying selection data in response to (i) the positional signal representing the position of said cursor corresponding to (ii) a position of a virtual object on said monitor;
  - defining virtual input and output type objects and associating with each a class of behavior responsive to said selection data;
  - defining a state table in response to said selection data including:
    - (i) states of a simulated system including the objects,
    - (ii) simulated events to be monitored,
    - (iii) simulated actions to be taken by the simulated system in response to the monitored events, and
    - (iv) next states of the simulated system to be entered in response to the monitored events; and
    - selectively displaying and manipulating said virtual objects in response to said state table, said selection data, other virtual objects and contents of variables.
6. The method of claim 5 further comprising the step of associating with ones of said virtual objects respective names of variables to which said associated class of behavior of said virtual objects are respectively responsive;
7. A method of prototype development and prototype simulation, comprising the steps of:
- defining graphic objects for display on a monitor;
  - associating with each of the graphic objects enumerated inputs to which each graphic object is responsive and a response of each graphic object to the respective enumerated inputs;
  - displaying a state table having plural rows and columns on the monitor;
  - receiving a graphic device input signal and, in response, positioning and displaying a cursor on the monitor;
  - detecting a position of said cursor corresponding to a position of one of said graphic objects on the monitor;
  - displaying a menu of input and responses associated with said one graphic object;

selecting an input and a response from the displayed menu; and  
entering data corresponding to the selected input and response in said state table.

8. The method of claim 7 further comprising the step of selectively displaying the graphic objects on the monitor in response to input and response data stored in said state table.

9. A method of graphically modelling a system, comprising the steps of:

defining a set of graphic objects and attributing to each

(i) at least one controlling parameter,

(ii) a set of one or more displayable visual characteristics, and

(iii) changeable features of said displayable visual characteristics responsive to said controlling parameter;

describing a state table defining the operation of a finite state machine, the state table including

(i) present states of the finite state machine,

(ii) events to be monitored in each of the present states,

(iii) controlling parameters to be generated in response to detection of a respective event to be monitored in said present states, and

(iv) next states of the finite state machine in response to detection of a respective event to be monitored in said present states;

operating said finite state machine responsive to said state table to display said graphic objects including

(i) defining an instant one of said present states of the finite state machine defined in said state table.

(ii) detecting an event of the events to be monitored in said instant present state,

(iii) generating a controlling parameter in response to said detected event of said instant present state in accordance with said state table, and

(iv) defining a next state of the finite state machine in response to said detected event of said instant present state in accordance with said state table; and

displaying said graphic objects in response to said controlling parameter generated by said generating step.

10. A system for developing and executing interactive visual applications wherein dynamic data is mapped into coherently animated virtual objects and operator interaction facilities are provided to interact with the virtual objects, the system comprising:

a central processing unit responsive to operator control signals for supplying address and control signals;

random access memory providing instructions to said central processing unit in response said address signals from said central processing unit;

a non-volatile storage medium for supplying said instructions to said random access memory in response to said control signals from said central processing unit;

a graphics display device responsive to said control signals from said central processing unit for displaying the virtual objects; and

operator input means for supplying said operator control signals to said central processing unit, said operator control signal including positional and alphanumeric signals, said operator input means including

(i) a locator/trigger device for providing said positional signals to the central processing unit, and

(ii) a data entry device for supplying said alphanumeric signals to said central processing unit;

said instructions including functional modules for defining operations of said central processing unit, including

(i) an object editor module responsive to said operator control signals for defining the virtual objects by relating a graphical appearance of said virtual objects to a behavior of said objects during execution by the system;

(ii) an integration editor module responsive to said operator control signals for mapping ones of the virtual objects into names of variables, the contents of memory locations of said random access memory corresponding to said named variables, said named variables being used for redrawing the virtual objects on the graphics display to reflect changes in the contents of the memory locations associated with said named variables and said named variables further supplying graphical access to data in response to said positional signals from said locator/trigger device and to screen locat ons of said virtual objects.

- (iii) an execution facility for redrawing ones of said virtual objects in response to dynamic data stored in said random access memory and in response to operator control signals, the execution facility further responsive to predetermined logic specification data;
- (iv) said logic specification data specifying sets of events and corresponding reactions which should occur in response to the respective events, an respective origin of the event, a state of the system wherein said state is a finite state machine model of the system, and predetermined conditions, wherein each of said events originating from one of (A) ones of said virtual objects, (B) a system clock signal, (C) control signals from said central processing unit, (D) messages from other computer systems, or (E) changes in the contents of the named variables;
- (v) said logic specification further specifying the next state that the system should transition to after an event is received;
- (vi) said logic specification further specifying the reactions that occur under control of the logic specification, ones of which affect ones of the virtual objects displayed on the display device; and a state table editor responsive to said operator control signals for defining said logic specification by specifying:
- (i) system states;
  - (ii) events and conditions to be recognized in respective ones of said states;
  - (iii) actions to be taken in response to respective ones of said events and states; and
  - (iv) next states to be entered in response to respective ones of said events and states.
11. The system of claim 10 wherein said logic specification is generated responsive to selection of the virtual objects displayed on the graphics display to fill in a state table including said events and conditions to be recognized and corresponding actions to be taken.
12. The system of claim 10 including means for specifying events and respective sources of said events by clicking with the locator/trigger device on the visual representation of the virtual object generating respective ones of said events.
13. The system of claim 12 wherein said state table includes said logic specification data.
14. The system of claim 13 including means responsive to said locator/trigger device for selectively inserting said logic specification data into said state table.
15. The system of claim 11 including means responsive to said operator control signals for entering state names of respective ones of said states into the state table.
16. The system of claim 15 wherein said means for entering said state names includes means for copying from one of said cells forming said state table to another of said cells forming said state table.
17. The system of claim 19 wherein said execution facility includes said logic specification and said execution facility is stored in said random access memory during a system execution mode of operation.
18. The system of claim 10 in which said logic specification is stored in a computer controlled setup forming a real equivalent of a modeled system, wherein said computer controlled setup exhibits a behavior substantially the same as a behavior of said modeled system.

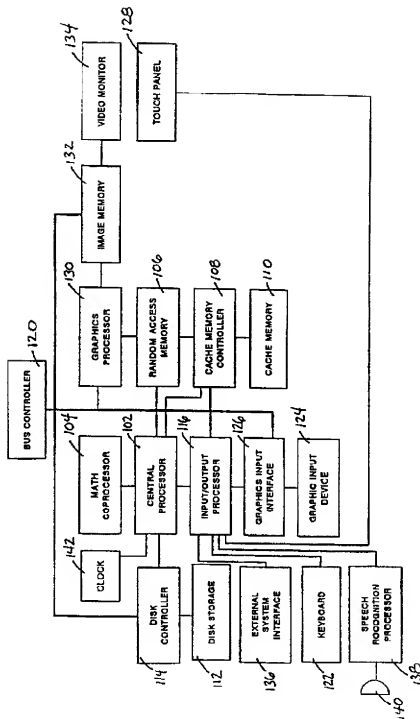


FIGURE 1

▲	STATE	EVENT	ACTION	NEXT STATE
▼				

Figure 2

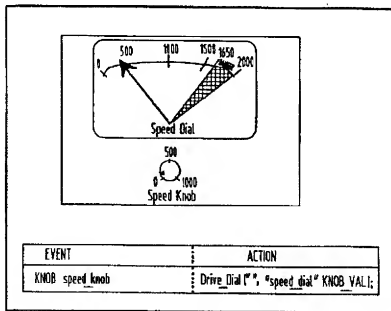


Figure 3

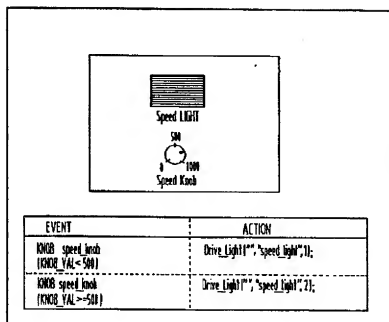


Figure 4

STATE	EVENT	ACTION	NEXT STATE
	KNOB speed_knob (KNOB_VAL < 500)	Drive_Light("", "speed_light", 1);	
	KNOB speed_knob (KNOB_VAL >= 500)	Drive_Light("", "speed_light", 2);	

Figure 5

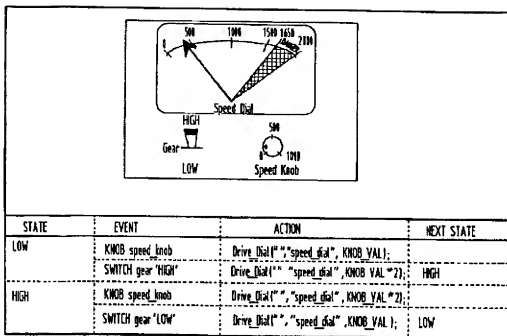


Figure 6

	STATE	EVENT	ACTION	NEXT STATE
OBJECT EVENTS		LOCATOR DOWN	/* This and the next two are events which can happen from LOCATORS	
		LOCATOR DRAG	/*	
		LOCATOR RELEASE	/*	
		FIELD	/* There are three variants of the FIELD event. These occur when the user types	
		FIELD VALID	/* In data to an input field */	
OTHER MOUSE INPUT		FIELD INVALID	/* This event occurs if you click on the background */	
		MOUSEMOVE	/*	
		ANYEVENT	/* Accept any event which comes your way. */	
		FIRST TIME	/* This event occurs the first time you enter this state. */	
		INITIALLY	/* This occurs when entering the state. */	
STATE TRANSITION EVENTS		IMMEDIATE	/* This occurs initially, and after every reaction. */	
		FINAL	/* This occurs if a transition to another state will occur. */	
		COMMENT	/* This is not event, just a comment. */	
		SERIAL	/* Input from a physical input device (in/dev). */	
		PERIODIC in Hz	/* This event occurs repeatedly every second. */	
TIME BASED EVENTS		PERIODIC in SECS	/* This event occurs repeatedly every few seconds. */	
		ON TIMEOUT in SECS	/* This event occurs repeatedly every few seconds. */	
		USER	/* User defined event. */	
		WHOMEVER	/* A pseudo event. */	

Figure 7



STATE	EVENT	ACTION	NEXT STATE
LOW	FIRST_TIME	Get_frame("control panel", 1, FOREGROUND_FRAME);	
	KNOB speed knob	Drive_Dial("speed_dial", KNOB_VAL);	
	SWITCH gear 'HIGH'	Drive_Dial("speed_dial", KNOB_VAL+2);	HIGH
HIGH	KNOB speed knob	Drive_Dial("speed_dial", KNOB_VAL+2);	
	SWITCH gear 'LOW'	Drive_Dial("speed_dial", KNOB_VAL);	LOW

Figure 8

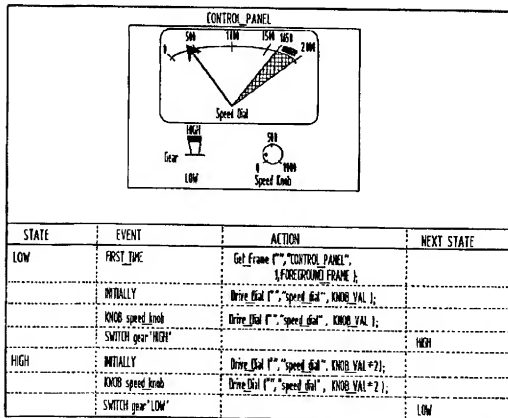


Figure 9

STATE	EVENT	ACTION	NEXT STATE
	FIRST TIME	Get Frame("","CONTROL_PANEL" 1, FOREGROUND_FRAME);	
	IMMEDIATE	fprint(dial.log,"%I % n, KNOB_VAL, POTEN_VAL);	
	KNOB speed knob 1	Drive Dial(...);	
	POTENTIOMETER speed knob 2	Drive Dial(...);	

Figure 10

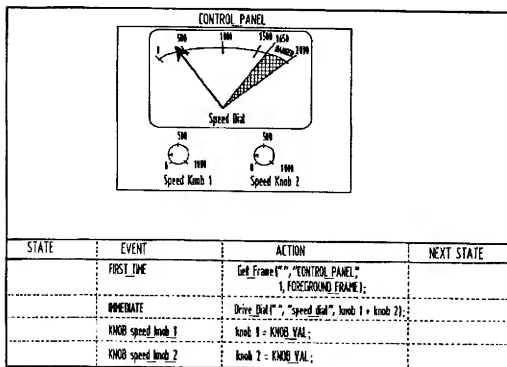


Figure 11

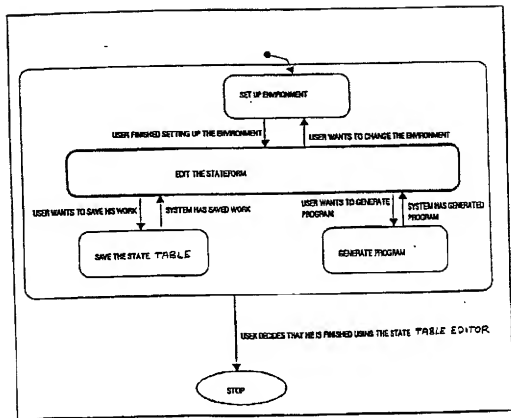


Figure 12

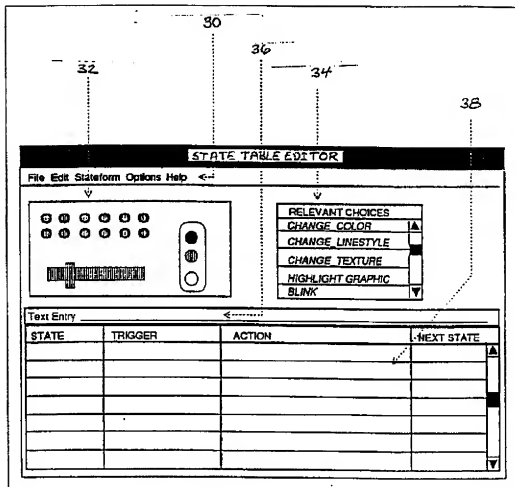


Figure 13

ACTION PARAMETERS	
Object: LIGHT / TrafficLight	
Action: Change State	
Name Of State:	RED_LIGHT

F Figure 14

ACTION PARAMETERS	
Object: XY OBJECT / Car Icon	
Event: Move_Object	
Anchor	RELATIVE_TO_CRT
X Coordinate	0.5
Y Coordinate	0.5

F Figure 15

Parameter Value
Field Name: NAME_OF_STATE
Object: LIGHT / TRAFFIC_LIGHT
Type: STATE_NAME
RED_LIGHT
GREEN_LIGHT
YELLOW_LIGHT
LEFT_TURN_LIGHT
WALK_LIGHT

Parameter Value
Field Name: ANCHOR
Object: XY OBJECT / Car Icon
Type: ANCHOR_TYPE
RELATIVE_TO_CRT
RELATIVE_TO_FRAME
ABSOLUTE
SCALED_TO_FRAME

Parameter Value
Field Name: X Coordinate      Object: XY OBJECT / Car Icon      Type: float
Enter Value (0.0,3.0) _____

Figure 16

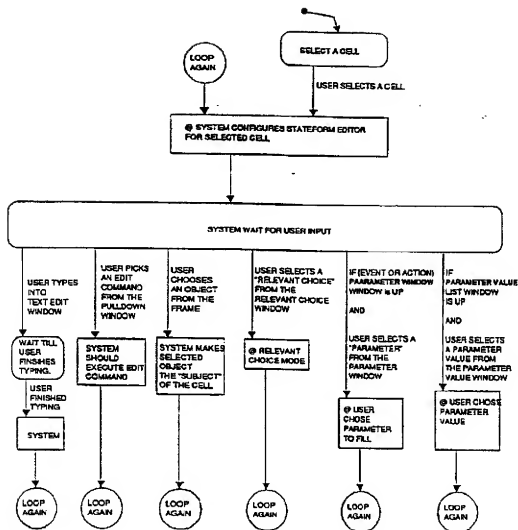


Figure 17

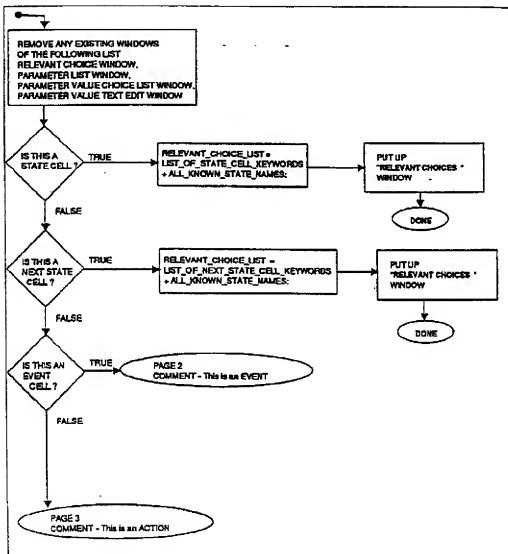


Figure 18

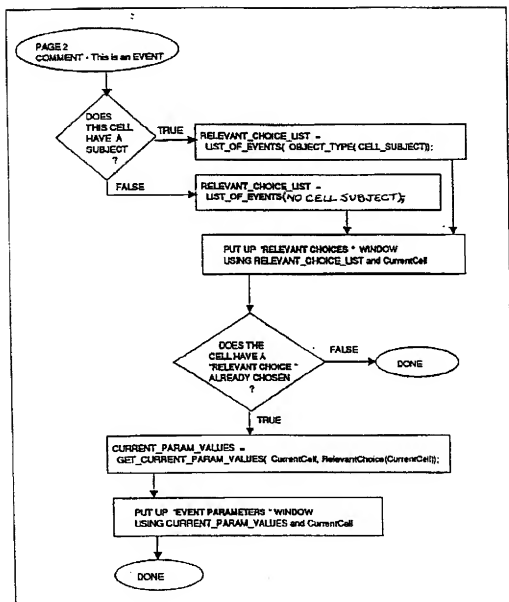


Figure 19



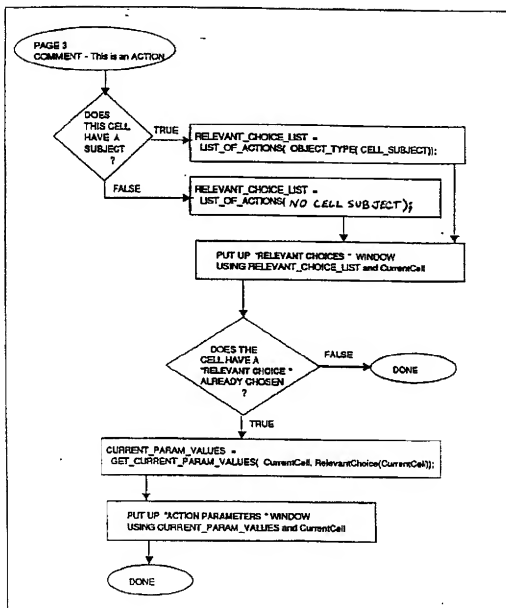


Figure 20

Examples of EVENTS for an object class:

Object Class: KNOB

KNOB\_TURNED  
KNOB\_MOVED  
KNOB\_PRESSED

Object Class: FIELD

FIELD\_VALID  
FIELD\_INVALID  
FIELD

Figure 21

Examples of EVENTS that apply to no object class:

Object Class: NONE

NO\_DEVICE  
ANYEVENT  
FIRST\_TIME  
INITIALLY  
IMMEDIATE  
FINAL  
COMMENT  
PERIODIC  
TIMEOUT  
USER

Figure 22

## Examples of ACTIONS for an object class:

Object Class: LIGHT

```

Light_Change_State      /* This changes the mode of the light. */
Object_Change_Back_Color /* This applies to all objects which have a
                        * background color. */
Object_Move             /* This applies to all movable objects. */

```

Object Class: FIELD /\* Text fields can be input, output, or both in and out \*/

```

Object_Change_Back_Color /* This applies to all objects which have a
                        * background color. */
Object_Move              /* This applies to all movable objects. */
Text_Font_Change         /* This applies to all objects that have text in
                        * them. */
Field_Set_Default        /* For a text field, set its default value. */
Field_Set_Validation     /* Set the validation rules for the field */
Field_Set_Value          /* Set the value of a field. */
Field_Restore_Default    /* Set the value of the field to the default
                        * value. */

```

Figure 23

## Examples of ACTIONS that apply to no object class:

Object Class: NONE

```

random_number          /* Compute a random number */
print_display          /* Send a copy of the display to the printer */
set_color_list         /* Add names to the color table. */
exit                  /* Stop all processing */

```

Figure 24

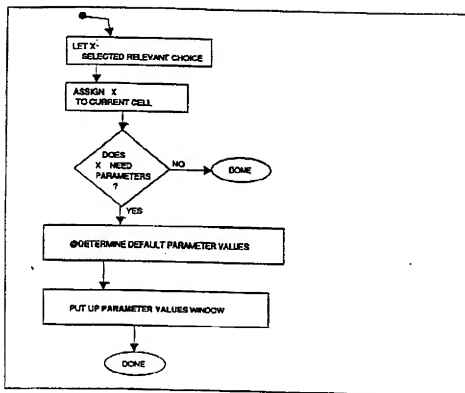


Figure 25

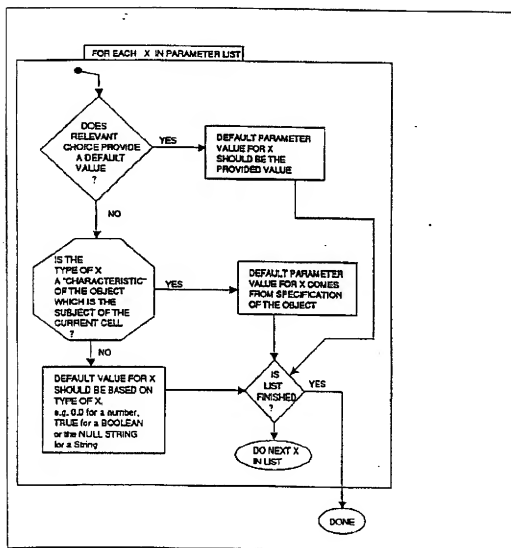


Figure 26

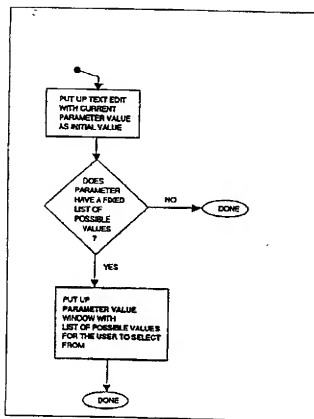


Figure 27

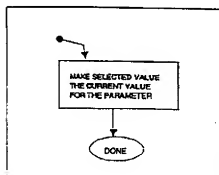


Figure 28

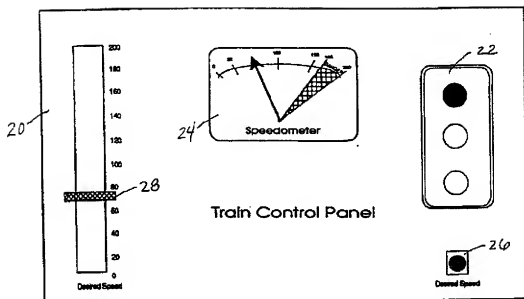


Figure 29

STATE	EVENT	ACTION	NEXT STATE
RED	FIRST TIME	Get_Frame("TRAIN", 0, FOREGROUND_FRAME);	
	INITIALLY	Drive_Dial("Speedometer", ZERO_VAL); Drive_Light("Traffic_Light", "RED");	
	BUTTON change_light		GREEN
GREEN	INITIALLY	Drive_Dial("Speedometer", POTEN_VAL); Drive_Light("Traffic_Light", "GREEN");	
	POTEN speed control		
	BUTTON change_light	Drive_Dial("Speedometer", POTEN_VAL);	YELLOW
YELLOW	INITIALLY	float desired_speed; Drive_Light("Traffic_Light", "YELLOW"); desired_speed = POTEN_VAL; if (desired_speed > 20.0) desired_speed = 20.0; Drive_Dial("Speedometer", desired_speed);	
	POTEN speed control	float desired_speed; desired_speed = POTEN_VAL; if (desired_speed > 20.0) desired_speed = 20.0; Drive_Dial("Speedometer", desired_speed);	
	BUTTON change_light		RED

Figure 30